▩ 연구논문

객체지향 ERP 시스템에서 데이터 객체 계층의 구축

Development of Data Object Layer (DOL) in Object-Oriented ERP Systems

김창욱 전진
Chang-Ouk Kim Jin Jun

ABSTRACT

To develop a generic ERP(Enterprise Resource Planning) system which can accommodate various types of manufacturing enterprises, object-oriented methods are commonly applied from analysis to implementation. The objective of OO-ERP (Object-Oriented ERP) systems is the reusability of business objects(components). In practice, one of the critical features for the reusable OO-ERP system would be the capability of interfacing with distributed, heterogeneous data repositories. Consequently, it is essential to provide data repository transparency in OO-ERP systems - business objects do not take care of the locations and types of data repositories. In this paper, we propose Data Object Layer(DOL) that supports such transparency. DOL is a horizontal component through which OO-ERP systems can be seamlessly connected with diverse data repositories.

1. 서론

제조기업들은 제조 및 경영 환경의 급진적인 변화에 대응하기 위하여 IT(Information Technology)를 이용한 경영 혁신을 시도하고 있다. 이러한 시도의 가시적인 결과는 경영 프로세스를 지원하기 위한 소프트웨어 시스템의 구축이나 도입으로 나타나고 있으며, 80년대의 MRP 에서 시작된 이 흐름이 CIM 등을 거치며 현재는 확장된 개념의 기업 관리 시스템인 ERP(Enterprise Resource Planning)의 구축 또는 도입으로 나타난다. 그러나 이러한 노력의 결과로 얻어지는 소프트웨어 시스템이 기업 입장과는 달리 변화하는 기업 경영 프로세스에 적절히 변화할 수 없기 때문에 '계속적인 경영 혁신(continuous business re-engineering)' 이라는 기업의 요구에는 부응하기 힘들다. 이러한 이유는 현재까지 소프트웨어의 제작방법으로 적용되던 구조적 개발 방법론 및 프로그래밍 언어의 한계로 인한 것이었다[4][8][9].

이러한 한계를 극복하기 위한 노력으로 컴퓨팅 환경의 새로운 방법론으로 제안되어 보급된 객체지향 방법론 및 이를 지원하는 프로그래밍 언어들이 출현하여 현실 반영 정도를 향상시키고, 소프트웨어의 유지 및 보수에 혁신을 가져올 것으로 기대되었다. 그러나 객체지향 기법을 적용하여 ERP 시스템을 구축하기 위해서는 몇 가지 이유로 어려움이 발생하였다. 그중 대표적

인 원인이 객체지향적 기법을 사용하여 구현된 응용 프로그래밍과 내부 객체들의 지속성(persistency)을 보장하기 위한 데이터 저장소(data repository)와의 인터페이스 작업이 어렵고 시간이 많이 걸린다는 점이다[3]. 즉, 경영 업무 어플리케이션은 데이터 저장소의 종류(관계형 데이터베이스/객체형 데이터베이스/객체관계형 데이터베이스 등의 DBMS의 종류, 실제 제품의 종류)와 네트워크 환경(클라이언트/서버 환경, 웹 환경, 통신 미들웨어), 구현 언어(C++, Java 등)에 따라서 각기 다른 형태와 방법으로 구현해야 하는 경우가 발생하고, 현실 상황에서는 대부분 개발에 사용되는 미들웨어를 포함한 개발환경에 종속되고 있다.

최근 들어, 소프트웨어 개발 관련 업계에서 컴포넌트(component)라는 단어를 많이 사용하기 시작했다. 즉, 기존의 객체지향적 접근 방법으로는 실제적으로 재사용 가능한 시스템을 구축할 수 없다는 인식 하에 독립적으로 실행가능하고, 재사용성이 높은 소프트웨어 덩어리라는 원칙에 따라서 구축된 컴포넌트를 개발하고 이를 바탕으로 어플리케이션 시스템을 개발하여야 한다는 것이다[6]. ERP의 경우, 특정 대상시스템으로만 국한하지 않고, 여러 시스템에 적용하기 위하여 객체지향 기법을 사용하는 경우가 많은데, 이러한 접근 방법도 결국은 컴포넌트를 개발하는 것으로 해석될 수 있다. 이러한 컴포넌트의 경우 독립적으로 실행가능하고 어느 시스템에서도 재사용되기 위해서는 컴포넌트에서 필요한 기능(예를 들어, 사용자 화면, 데이터 저장소 접근 기능 등)을 특정 네트워크 구조나, 데이터 저장소의 종류, 구현 언어에 무관하게 사용할 수 있어야 한다. 특히 위에서 언급한 데이터 저장소와의 인터페이스는 이러한 투명성(네트워크, 데이터 저장소 종류, 구현언어) 문제를 해결해야 하는 중요한 기능이다. 본 연구에서는 ERP 시스템이 이와 같은 특징을 가지는 데이터 인터페이스를 구축하기 위하여 다음과 같은 해결방법을 제안하고자 한다.

- 데이터 객체(data object)라는 개념을 정의하고 이를 관리하는 데이터 객체 계층(data object layer)을 두어, 어플리케이션 계층과 실제 데이터 접속 기능 미들웨어(예, ODBC, JDBC 등)의 분리를 이루고자 한다.
- 또한 어플리케이션 계층과 데이터 객체 계층을 분산 객체 미들웨어(예, CORBA, DCOM 등)로 연결할 수 있도록 하여, 어떠한 네트워크 구조(2단계 구조, 3단계 구조, 웹 기반 등)에도 무관한 어플리케이션 프로그램을 작성할 수 있도록 한다.

이때, 어플리케이션 계층에서 바라보는 데이터 객체와 물리적 데이터 저장소에서 바라보는 데이터 사이의 개념적 차이를 극복하기 위한 장치로는 권일명 등[1]에서 제안한 '일반화 패턴'을 사용하여 일반화된 데이터 객체를 관리함으로써 특정 데이터 객체마다 인터페이스 프로그램 작업을 하지 않아도 되도록 자동화하였다.

2. 기존 연구

본 연구에서 정의하고 있는 데이터 객체는 그림 1에서 보는 바와 같이 어플리케이션과 데이터 저장소 사이에 존재한다. ERP 시스템의 업무처리에 해당하는 비즈니스 객체(business object)의 기능 중 데이터 관련 기능을 분리하여 데이터 객체와 함께 집중한 것이 데이터 객체 계층 (DOL : Data Object Layer)이다(그림 1). 이와 같이 데이터 객체와 데이터 관련 기능을 비즈니스 객체에서 분리하여 독립적 계층(DOL)으로 집중하는 이유는 1) 어플리케이션과 데이터 저장소 사이의 투명성을 높이고, 2) 데이터 관련 요청에 대한 동시 제어(concurrency control)와 비즈니스 트랜잭션(business transaction)의 문제를 해결하기 위해서이다.

객체지향 기법을 이용하여 구축되고, 실제 사용되는 대형 어플리케이션의 경우 대부분 데이터 객체의 필요성을 인정하고 있다. 객체지향 기법을 이용하여 실제 구축된 시스템의 사례를 분석한 논문[15]에서는 이러한 데이터 객체의 성격을 가지는 객체를 정의하고 있으며, 실제 객체지향 시스템의 대부분이 이러한 객체의 구축에 초점을 두고 있다고 서술하고 있다. 그러나, 이 논문에서도 데이터 객체를 관리하는 기능에 대한 구체적인 언급은 없다.

김창욱 등[2][5]은 객체지향적 분석, 설계, 구현 각 단계에서 데이터 객체의 필요성과 그 정의를 서술하고 있으며, 데이터 객체와 데이터 저장소 사이의 데이터 베이스 접속 기능을 담당하는 객체가 필요함을 강조하였고, 이러한 데이터 베이스 접속 기능은 권일명 등[1]의 논문에서 구축되었다.

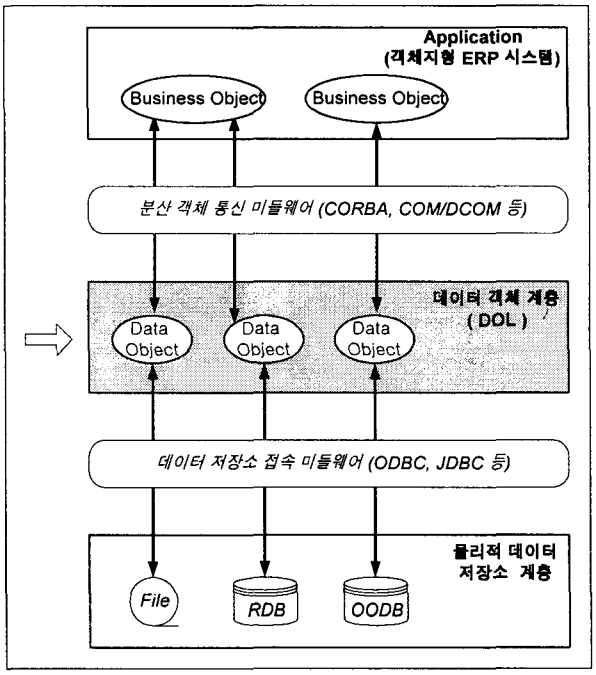

그림 1. 데이터 객체 계층의 구조

데이터 저장소와 어플리케이션의 분리를 위한 접근 방법으로는 ODBC[11], JDBC[14] 등의 기존 데이터 베이스 접속 미들웨어를 이용하는 방법과 데이터 저장소와 데이터 객체를 논리적으로 연결하여 데이터 저장소의 형태에 무관한 객체지향 어플리케이션 작성을 가능하도록 하는 방법이 있다.

두 가지의 접근 방법 중 후자의 방법인 데이터 저장소와 데이터 객체를 연결하는 데이터 인터페이스의 구현에 관한 연구는 현재까지 많이 진행되어 왔다. 기존의 연구들은 대부분 데이터 객체를 보존하는 데이터 저장소로써 관계형 데이터 베이스를 이용할 때 발생하는 문제를 설명하고, 이의 해결을 위한 방안의 연구에 중점을 두었다. Ivar Jacobson[9]은 지속성이 있는 데이터 객체의 저장소로 관계형 데이터 베이스를 사용할 때 발생할 수 있는 임피던스 문제(impedance problem)로 지적하였고, Kyle Brown등[10]의 연구에서는 이러한 임피던스 문제를 해결하기 위한 데이터 객체 및 데이터 테이블의 설계방안을 패턴 언어[7]를 통해 설명하였다. 또한 Arthur M. Keller등[3]과 Shailesh Agarwal등[16]은 객체지향적 데이터 객체의 특성을 지원하기 위한 데이터 테이블의 설계방안 및 임시 저장소를 사용하는 구조를 도입하였다. 권일명 등[1]은 각 데이터 객체별로 데이터 인터페이스를 구현하는 경우 데이터 객체의 종류에 따라

인터페이스의 구축에 많은 시간이 요구되는 점을 지적하였다. 이를 해결하기 위하여 각 데이터 객체별 개별 형태를 일반적인 형태로 변형하는 '일반화 패턴'을 제안하여 인터페이스 구축의 자동화를 이루었다. 그러나 이러한 연구들은 주로 데이터 저장소와 데이터 객체 사이의 인터페이스 문제, 즉 임피던스 불일치와 구현의 자동화에 초점을 두고 있어서 어플리케이션이 데이터 객체를 이용하는 데 있어서 고려해야 인터페이스 기능에 대한 연구는 부족하다.

ODBC와 JDBC와 같은 기존 데이터 베이스 접속 미들웨어는 데이터 저장소의 종류와 위치에 무관한 어플리케이션 작성을 위하여 정의되었다. 특히 JDBC는 Java를 기반으로 구현되어 ODBC의 단점이었던 통신상의 보안 문제와 웹 기반 클라이언트/서버 환경에서의 구현문제를 해결하였다. 그러나, JDBC의 경우에도 데이터 베이스 접속 기능을 담당하는 JDBC 드라이버에서 사용하는 네트워크 프로토콜에 의하여 네 가지 종류로 분류되고 그 드라이버 종류에 따라서 2 단계, 3 단계의 네트워크 구조를 가진다[14]. 이러한 이유로, 객체지향 ERP 시스템을 구축하는 경우에 JDBC를 사용하더라도 네트워크 프로토콜과 데이터 베이스(네트워크/Java 지원), 클라이언트/서버 환경(웹 기반 여부)에 따라서 어플리케이션 작성이 변하게 된다.
기존 연구와 규약들에 대한 문제점을 정리하면 다음과 같다.

- 어플리케이션이 데이터 저장소와 네트워크 구조, 구현 언어 등으로부터 완전히 은닉하지 못하기 때문에 어느 하나라도 변경되면 어플리케이션이 이에 맞춰 변경되어야 하므로 특정 환경에 맞게 구현된 어플리케이션(객체지향 ERP 시스템)은 다른 환경에서는 그대로 사용할 수 없다.

본 연구에서는 위와 같은 문제점을 해결하기 위하여 데이터 객체 계층을 두어 데이터 저장소와 어플리케이션 사이에 완전한 투명성(transparency)을 구현하고자 한다. 본 논문의 3장에서는 객체지향 ERP 시스템에 대한 설명을 통하여 데이터 객체의 필요성과 일반화 패턴을 통한 데이터 객체의 일반화, 데이터 객체 계층의 구조, 그리고 기능을 제안한다. 그리고 제안된 구조를 사용하여 어플리케이션이 네트워크, 데이터 저장소, 구현 언어에 대한 투명성을 가질 수 있음을 4장에서 보인다. 마지막으로 5장에서는 결론 및 추후 연구 분야에 대해 서술한다.

3. 데이터 객체 계층

3.1 객체지향 ERP 시스템

순수한 형태의 객체지향 시스템을 설계, 구현하는 데 있어서는 몇 가지 문제점이 있다. 김창욱 등[2][5]은 개발될 시스템의 재사용성을 높이기 위하여 분석-설계-구현 각 단계에서 객체지향 기술을 적용하였다. 객체지향 기술의 핵심은 데이터와 그 데이터에 관련된 함수를 한 객체 내에 두는 것이라 할 수 있으나, 적용 과정에서 이들은 ERP 시스템과 같이 기능이 강조가 되고 여러 기능이 동시에 한 데이터를 접근하는 형태의 대형 시스템의 경우에는 데이터 은닉(encapsulation)과 같이 객체지향 기법이 지향하는 형태로 객체를 설계하기에 문제점이 있음을 지적하였다. 이러한 문제점들을 정리해 보면 다음과 같다.

- 각 객체들 사이에 많은 상호참조(interaction)가 존재하여, 객체지향의 장점인 black-box 접근방법을 이루기 어려워지고, 결국 객체 설계 시 다른 객체의 내부를 들여다 봐야하는 white-box 접근방법으로 갈 수밖에 없다.
- 분석 단계에서 추출된 각 객체들의 지속성을 보장하기 위한 인터페이스 작업에 많은 양

의 코드와 시간이 소요된다.

- 각 객체 내의 정보를 접근하는 외부의 많은 기능 객체들이 있을 경우, 이들 간의 동시 제어가 필요하다.

위의 문제점 중 첫 번째 문제점에 대해서는 객체지향 연구자, 개발자들도 인식하고 있으며, 대안으로 소프트웨어 컴포넌트(software component)를 제안하고 있다. 소프트웨어 컴포넌트는 기존의 객체지향 기법에서 추구하던 black-box 접근방법을 해결한 접근방법이며, 코드 수준이 아닌 실행파일 수준의 재사용을 통한 소프트웨어 생산성을 높이기 위한 노력이라고 볼 수 있다[6].

반면, 객체 내부 데이터 관리는 컴포넌트의 접근방법으로 해결되지 않는 다른 영역의 문제인데, 특히 객체의 지속성, 즉 데이터 저장소를 이용하여 객체 내부 데이터의 지속성을 구현하는 문제는 중요한 문제로 인식되었고, 많은 연구들과 제품들의 성과가 있었다. 또한 여러 객체들 간의 상호참조(interaction)가 많아짐에 따라 객체 내부의 정보를 접근하려는 많은 트랜잭션 사이에서 동시 제어할 수 있는 기능 또한 그 필요성이 인식되고 있다.

김창욱 등[2][5]의 논문에서는 이러한 객체 내부 데이터 관리에 대한 문제점을 인식하여 시스템 분석 단계에서부터 물리적 객체, 데이터 객체, 기능 객체를 분리하고, 이렇게 분리된 객체들을 기반으로 설계 단계에서는 여러 기능 객체에서 참조하는 데이터 객체들의 경우에는 독자적으로 분리하고 데이터 객체화하여 위의 문제점을 해결할 수 있는 분석방법론을 제안하고 있다(그림 2). 데이터 객체와 기능 객체를 물리적 객체에서 분리하는 접근방법은 데이터 은닉(encapsulation)과 같은 객체지향 특성과는 거리가 먼 형태이지만, 위에서 지적한 문제를 해결하기 위해서는 불가피하다는 점을 그의 논문에서 지적하였다. 또한, 같은 논문에서 데이터 객체와 데이터 저장소 사이의 인터페이스 기능을 담당하는 객체군을 정의하고 있으며 이러한 데이터 객체와 데이터 저장소 사이의 인터페이스 구현에 관한 내용은 권일명 등[1]의 논문에서 다루고 있다.

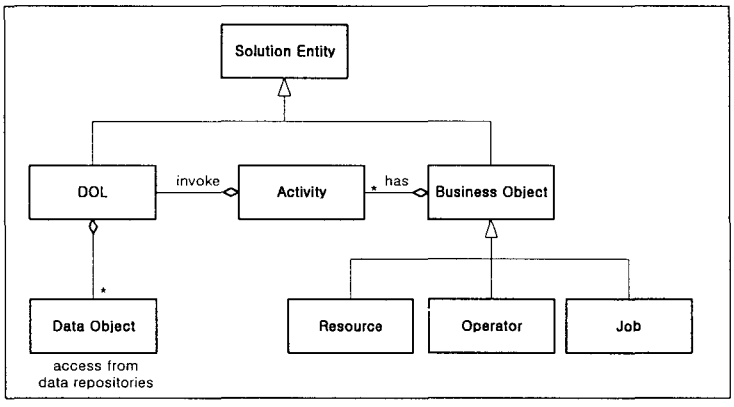

그림 2. 생산 시스템에 대한 설계모델

정리하자면, 객체지향 기법을 ERP와 같은 대형 시스템에 적용하여 실제적으로 구축하기 위해서는

1. 데이터 객체의 개념이 필요하며,
2. 데이터 객체와 데이터 저장소 사이의 인터페이스 기능을 담당하는 객체가 필요하다.

이러한 필요성을 바탕으로 '일반화 패턴'을 이용한 데이터 객체 표현과 데이터 저장소 인터페이스에 대하여 다음절에서 서술하고자 한다.

3.2 데이터 객체 (Data Object)

어플리케이션(객체지향 ERP 시스템)에서 사용하는 데이터 객체는 주문 데이터 객체, 고객 데이터 객체, 제품 데이터 객체 등 많은 종류가 있다. 이들은 각 데이터 객체별로 고유한 모양, 즉 개별 뷰(specific view)를 가지고 있는데, 각 데이터 객체의 개별 뷰는 그 객체가 가지고 있는 속성(attribute)들인 내부 변수(member variable)들에 의해 규정되어 진다. 즉, 각각의 다른 데이터 객체에 들어있는 내부 변수 개개의 특성을 일반화할 수 있다면 이들의 조합인 데이터 객체의 일반화도 이룰 수 있다는 것이다. 이러한 '일반화 패턴(generalizing pattern)'을 이용하여 개별 데이터 객체를 공통 형태의 데이터 객체로 뷰를 전환한다면 데이터 저장소와의 인터페이스는 개별 데이터 객체별로 각각 구축하지 않고 공통 형태로 인터페이스를 구축할 수 있게 되므로 인터페이스 작업의 자동화를 이룰 수 있다.

3.2.1 일반화 패턴(generalizing pattern)

각기 다른 데이터 객체는 내부 변수의 종류와 수에 의해서 그 특성이 결정된다. 일반화 패턴에서 이러한 내부 변수의 일반형을 추출하여 개별 데이터 객체 내의 내부 변수를 일반형으로 표현한다. 그림 3는 데이터 객체인 주문 데이터 객체(OrderData)를 OrderID, CustomerID, Duedate, Price 등의 객체 내부 변수의 조합(표 1)으로 보고 있으며, 각 내부 변수들을 일반형 객체(Var)로 표현하여 주문 데이터 객체는 네 개의 Var 객체의 조합으로 표현이 되고 있다. 또한 각각의 Var 객체는 주문 데이터 객체의 내부 변수 주소와 형(type)정보를 가지고 있으므로, Var객체를 GenericData라는 일반형 데이터 객체에서 포함(aggregation)하게 한다면 OrderData 객체는 GenericData 객체로 표현되는 일반화된 객체가 된다(그림 4).

이러한 구조를 이용하면 모든 데이터 객체는 일반화되어 GenericData형으로 변환될 수 있으므로, 데이터 객체와 데이터 저장소 사이의 인터페이스는 일관적인 형태로 표현될 수 있다. 그러므로, 어플리케이션에서는 각기 다른 형태의 실제 데이터 객체를 사용하고, 데이터 저장소와의 인터페이스는 공통적인 형태를 가진 GenericData객체를 통해 실제 데이터 객체를 관리할 수 있다. OrderData 객체가 생성될 때 각 내부 변수의 일반화를 위한 Var형 객체를 생성하고, 이를 register 내부 함수를 통해 표 2와 같이 등록하면, 기초 클래스의 객체인 GenericData형 객체가 Var형 객체를 보유하게 되어 자식 클래스의 객체인 OrderData형 객체의 각 내부 변수를 접근할 수 있게 된다. 또한 Var형 객체는 데이터 객체의 내부 변수를 접근할 수 있어야 하기 때문에, 생성시점에서 내부 변수의 주소를 인자로 받아들인다. 이러한 각 내부 변수와 데이터 저장소간의 관계는 메타 데이터(meta data)로서 표현되어 저장된다.

표 1에서 EXPR_OrderID등의 Expr형 객체는 질의 생성을 위한 멤버 변수로써 데이터 객체를 통한 데이터 저장소 접근 요청 시 이용된다. 즉, 데이터 저장소로 요청을 하는 경우 >=등의 연산자와 상수 값은 모두 Expr형 객체로 변화되어지고, 실제 데이터 저장소에게는 이들 Expr 객체들의 조합의 형태로서 요청(query)하게 된다.

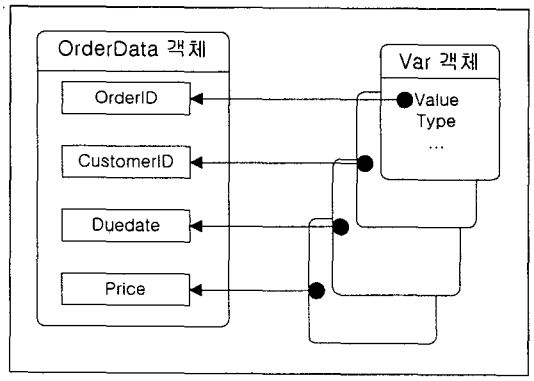

그림 3. OrderData 객체의 내부 변수와 Var 객체간의 관계

표 2 OrderData 클래스의 선언

```
class OrderData : public GenericData
{
private:
   char OrderID[20];
   char CustomerID[25];
   date Duedate;
   long Price;

   // Query 생성을 위한 내부 변수
   Expr EXPR_OrderID;
   Expr EXPR_CustomerID;
   Expr EXPR_Duedate;
   Expr EXPR_Price;
public:
   OrderData();
   ~OrderData();
   ...
};
```

표 3 OrderData 클래스 생성자의 정의

```
OderData::OrderData()
{
   Var* aVar;
   // 내부 변수 OrderID의 일반화
   aVar=new Var("Order","OrderID",&OrderID);
   register(aVar);
   EXPR_OrderID = aVar->retExpr();

   //나머지 내부 변수 처리도 위와 같은 형태임
}
```

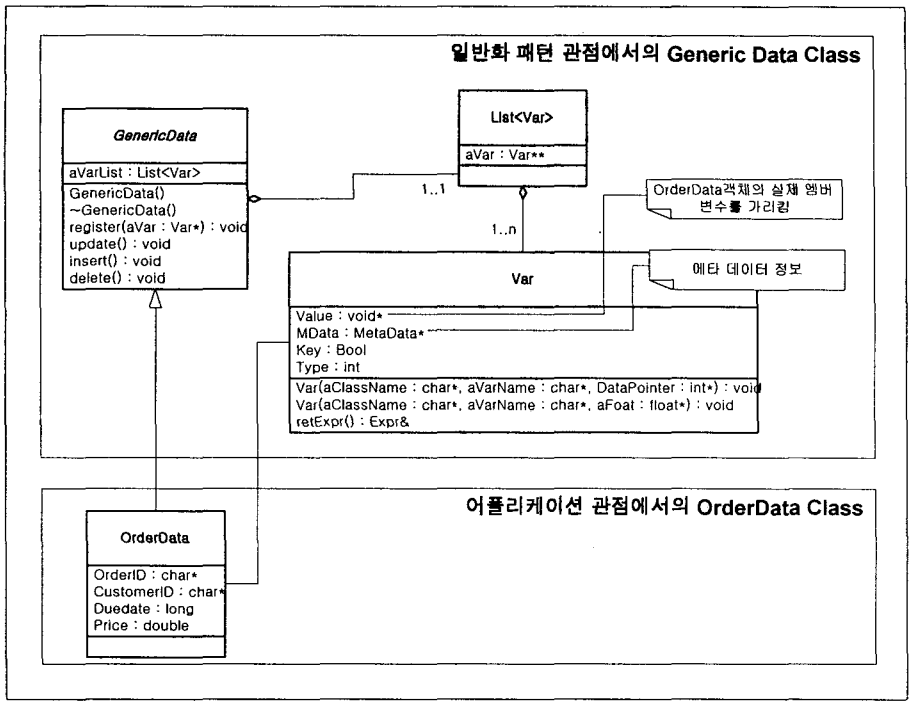

그림 4. 데이터 객체의 일반화를 위한 일반화 패턴의 구조

3.2.2 메타 데이터(meta data)

Var 객체는 데이터 객체의 내부 변수에 대한 참조 정보와 함께 내부 변수와 데이터 저장소 간의 관계 데이터, 즉 메타 데이터를 관리한다. 메타 데이터는 데이터베이스나 파일에 저장되어 있으며 그림 5와 같이 클래스 명과 내부 변수 명을 기본 키(primary key)로 하는 테이블의 형태이다. 그림 5에서는 Order클래스와 Product클래스의 메타 데이터를 보여주고 있다. 이러한 메타 데이터를 읽어들이기 위해 Var형 객체는 생성 시 메타 데이터의 기본 키 정보인 클래스 명과 내부 변수 명을 인자로 받아들여(표 2), 필요한 메타 데이터를 메타 데이터 저장소(데이터 베이스나 파일)에서 획득하게 된다. 메타 데이터가 어플리케이션으로부터 분리되어 있는 구조는 데이터 저장소의 위치 및 데이터 스키마 등 데이터의 구조가 변경되면 관련된 메타 데이터만 고치면 되므로, 데이터 객체를 사용하는 어플리케이션은 변경에 따른 영향을 적게 받을 수 있다.

Mapping Data : 테이블

클래스 명	멤버변수 명	데이터 저장소 종류	서버	데이터베이스 명	테이블 명	필드명
Order	CusID	ODBC	Server1	OrderMgt.mdb	OrderTable	CusID
Order	dDate	ODBC	Server1	OrderMgt.mdb	OrderTable	DueDate
Order	Description	ODBC	Server1	OrderMgt.mdb	OrderTable	Description
Order	OrderedDate	ODBC	Server1	OrderMgt.mdb	OrderTable	OrderedDate
Order	OrderID	ODBC	Server1	OrderMgt.mdb	OrderTable	OrderID
Order	Price	ODBC	Server1	OrderMgt.mdb	OrderTable	Price
Product	ProductID	File	Server3	Product.dat	Product	ProductID

그림 5. 메타 데이터 테이블

3.3 데이터 객체 계층 (Data Object Layer)

본 절에서는 앞 절에서 정의한 데이터 객체를 관리하기 위한 데이터 객체 계층을 정의하고 이 계층의 기능들에 대해서 설명하고자 한다. 데이터 객체 계층 (DOL : Data Object Layer)의 기능들을 정리해보면 다음과 같다.

- 어플리케이션 객체(기능 컴포넌트)와 데이터 객체 사이의 연결 기능
- 데이터 객체를 관리함으로써 이중 데이터 저장소 화
  - 여러 기능 컴포넌트에서 접근하여 동일한 데이터를 변경하려는 경우, 동시 제어
  - 임시 저장소를 이용하여 요청 처리속도를 향상
- 데이터 객체와 데이터 저장소 사이의 인터페이스 기능을 담당하는 미들웨어(ODBC, JDBC, 그 밖의 데이터 베이스 회사에서 제공하는 데이터 베이스 드라이버 등)를 관리하는 기능

위와 같은 기능을 담당하기 위해서 그림 1과 같은 구조의 데이터 객체 계층을 제안한다. 그림에서 보면 어플리케이션과 데이터 객체 계층 사이는 CORBA[12]와 같은 분산 객체 미들웨어를 사용하여 연결하고 있다. 이러한 CORBA 또는 DCOM과 같은 분산 객체 미들웨어는 IDL(Interface Definition Language) 등의 인터페이스 언어를 제공하여 클라이언트와 서버의 구현언어, 플랫폼 등에 무관하게 정의할 수 있도록 도와준다. 즉, 어플리케이션에 해당하는 기능 컴포넌트와 데이터 객체 계층 사이에 이러한 분산 객체 미들웨어로 연결함에 따라, 사용 플랫폼, 구현언어에 대한 투명성을 보장한다.

데이터 객체 계층이 가지고 있는 두 번째 기능으로는 데이터 객체를 관리하는 기능이다. 여기서 담당하는 기능으로는 데이터 객체를 접근하는 기능 컴포넌트가 많은 경우, 이들의 데이터 접근 트랜잭션을 처리하는 기능으로서 동시 제어의 기능을 담당하게 된다. 또한 데이터 객체 계층이 메모리 위에 상주하면서 데이터 저장소에서 읽어들였던 데이터를 임시 저장소에 저장하여 기능 컴포넌트로부터 받은 요청을 처리할 때 처리속도를 높일 수 있다.

그 밖의 기능으로는 실제 데이터 저장소와 데이터 객체를 연결해주는 인터페이스 컴포넌트(ODBC, JDBC 등)들을 등록하여 사용할 수 있도록 관리하는 기능이 있다.

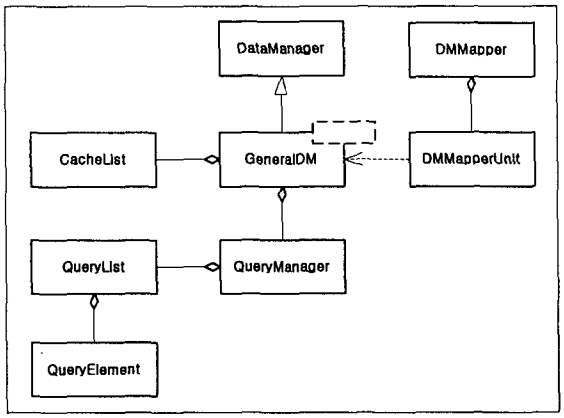

그림 6. 데이터 객체 계층의 객체 관계도

데이터 객체 계층은 데이터 객체 관리자 객체군과 이러한 관리자 객체를 어플리케이션에서 사용할 수 있도록 연결해 주는 객체(DMMapper)로 크게 분류된다. 데이터 객체 관리자 객체군

은 다시 실제 데이터 객체의 관리 객체인 DM 객체(Data Manager object)와 데이터 요청을 분석하여 처리하기 위한 객체군으로 나뉘어진다. 그림 6은 데이터 객체 계층 내의 객체들의 관계를 보이고 있다. 이러한 DM 객체는 C++언어의 기능인 템플레이트(template) 기능을 사용하여 특정 데이터 객체를 전담 관리하는 DM 객체, 예를 들어 주문 데이터 객체용 DM 객체를 따로 만들지 않고, 데이터 객체들을 '일반화 패턴'[1]을 이용하여 일반적인 관리 기능을 사용할 수 있도록 하여, 인터페이스 기능 구현의 자동화를 이루고 있다. 본 연구에서는 이러한 DM 객체를 GeneralDM으로 정의하여 여러 형태의 데이터 객체를 관리할 수 있는 일반적 기능의 데이터 객체 관리자 객체로 설계하였다. 또한 이러한 관리자 객체는 간단한 기능의 임시 저장소를 두고 있어 어플리케이션으로부터 받는 요청을 기존 임시 저장소의 내용과 비교하여 없는 경우에만 데이터 저장소에 접근하도록 지시하여 성능을 향상시킬 수 있도록 하였다.

3.3.1 GeneralDM 객체

관리자 객체(GeneralDM)가 관리해야 하는 데이터 객체는 그 특성에 따라서 동적 데이터 객체와 정적 데이터 객체로 나누어진다(그림 7). 동적 데이터 객체의 경우, 이를 사용하는 어플리케이션에서 참조, 수정, 삭제, 생성, 추가 등 데이터에 관련된 모든 기능이 필요한 반면, 정적 데이터 객체의 경우에는 어플리케이션에서 필요로 하는 대부분의 기능이 참조(select) 기능으로 한정된다. 특히 정적 데이터 객체의 경우에는 데이터 변경의 권한이 대부분 관리 부서에만 있기 때문에 부서별로 접근할 수 있는 기능을 제한하는 기능을 가지고 있다.

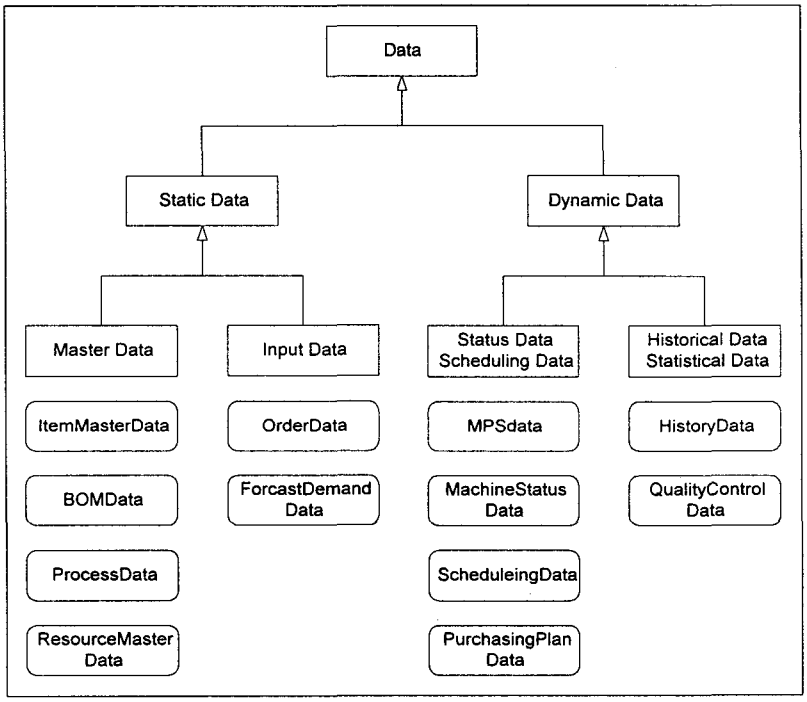

그림 7. 데이터 객체의 종류

이와 같이 다른 특성을 가진 데이터 객체를 관리하기 위하여 GeneralDM을 생성시점에서 정적 데이터 객체용 GeneralDM과 동적 데이터 객체용 GeneralDM으로 분류되어 생성된다(표 4). 각 데이터 객체마다 다른 GeneralDM에서 필요로 하는 내부 정보를 만드는 것은 C++의 '#define' 선언문을 이용하여 'DECLARE_GENERAL_DM(자신이 만든 데이터 객체명)'과 같은 방법으

로 사용된다(표 3). 표 3에서 보듯이 하나의 데이터 객체마다 _DataListRepository 라는 임시 저장소용 리스트 객체와 _QueryHistoryRepositroy 라는 요청정보를 보관하여 추후의 요청 시 기존 요청정보와 비교하여 임시저장소를 이용할 지를 결정할 수 있게 해주는 리스트 객체가 생성된다. 이들 객체는 GeneralDM 객체 내부에서 관리된다.

표 4. GeneralDM define code

```
#define DECLARE_GENERAL_DM(className)    \
        DBDoubleListPtr<className>* GeneralDM<className>::_DataListRepository    \
                = new DBDoubleListPtr<className>;    \
        QueryManager<className>*    GeneralDM<className>::_QueryHistoryRepository    \
                = new QueryManager<className>;
```

표 5. 정적 데이터 관리자 객체, 동적 데이터 관리자 객체의 생성

```
// 정적 데이터 객체(itemData)의 관리자 객체 생성
        GeneralDM<itemData> *StaticItemManager
                = new GeneralDM<itemData>(GeneralDM<itemData>::STATIC);

// 동적 데이터 객체(schedulingData)의 관리자 객체 생성
        GeneralDM<schedulingData> *DynamicSchedulingManager
                = new GeneralDM<schedulingData>(GeneralDM<schedulingData>::DYNAMIC);
```

표 6. GeneralDM의 Select, Update, Delete, Insert 예

```
        DBList<ItemData> list = new DBList<ItemData>;
        ListIterator Iterator = new ListIterator(list);

        //-----------------------------------------
        // Select :
        //-----------------------------------------

        StaticItemManager->getData(list, temp.SELitemID() > "item200" &&
                        temp.SELitemID() <= "item210" &&
                        temp.SELname() != "part1011" );

        //-----------------------------------------
        // Update :
        //-----------------------------------------
        pTemp = (itemData *)Iterator();
        pTemp->itemID = "updatedeID";
        pTemp->name = "part1011";
        pTemp->description = "갱신됨";
        pTemp->setStatus(DBUpdate);

        //-----------------------------------------
        // Delete :
        //-----------------------------------------
        pTemp = (itemData *)Iterator();
        pTemp->setStatus(DBDelete);

        //-----------------------------------------
        // Insert :
        //-----------------------------------------

        pTemp->itemID = "insertedID";
        pTemp->name = "part2012";
        pTemp->description = "추가됨";
        pTemp->setStatus(DBInsert);
        list.insert(pTemp);

        //-----------------------------------------
        // Update, Delete, Insert를 한번에 Setting
        //-----------------------------------------
        StaticItemManager->setData(list);
```

GeneralDM 객체의 사용 예는 표 4와 표 5이다. 표 4는 GeneralDM의 정적 데이터 객체의 일종인 itemData 객체를 관리하는 정적 관리자 객체와 동적 데이터 객체의 일종인

schedulingData 객체를 관리하는 동적 관리자 객체의 생성을 예를 들어서 보이고 있다. 표 5는 데이터 인터페이스 기능인 select 기능에 해당하는 **getData()** 함수와 update, delete, insert의 기능에 해당하는 **setData()** 함수의 예를 보이고 있다. **getData()** 함수는 그 입력 인자로서 Expr 형 객체의 조합을 받아들여 이를 데이터 저장소에 쿼리(query)형태로 넘겨주게 된다. 또한 **setData()** 함수의 경우에는 update, delete, insert 기능을 그 시점에서 처리하는 것이 아니라 각각 **DBUpdate**, **DBDelete**, **DBInsert**라는 상태 변수로 지정해 두었다가 **setData()** 함수에서 일괄적으로 처리하도록 설계되어있어 편리함을 제공하고 있다.

3.3.2 DMMapper 객체

DMMapper 객체는 어플리케이션에서 GeneralDM 객체를 사용하려고 할 때, 필요한GeneralDM 객체를 접근할 수 있도록 도와주는 객체이다. 즉, GeneralDM 객체의 생성 시 DMMapper 객체에게 생성되었음과 위치를 알려주어 한번 시작된 GeneralDM 객체에 대해서는 DMMapper가 어플리케이션에게 그 객체의 접근위치만을 넘겨준다.

정적 데이터 객체를 관리하는 GeneralDM의 경우는 CORBA 등의 분산 객체 미들웨어를 이용하여 인터페이스 객체의 주소를 DMMapper에 등록한 후 사용할 수 있도록 되어 있다. 동적 데이터 객체를 관리하는 GeneralDM의 경우는 기능 컴포넌트가 필요한 GeneralDM을 생성하여 DMMapper에 갖고 있다가 필요한 경우 GeneralDM을 얻어서 사용하고 있다. 한 기능 컴포넌트에서 사용이 끝난 GeneralDM의 소멸자를 부를 경우(delete)에는 DMMapper에 있는 등록 내용을 자동삭제하며, 필요 GeneralDM이 미등록 되어 있다든지, 중복되어 등록되려 할 경우 ERROR로 처리하도록 하고 있다.

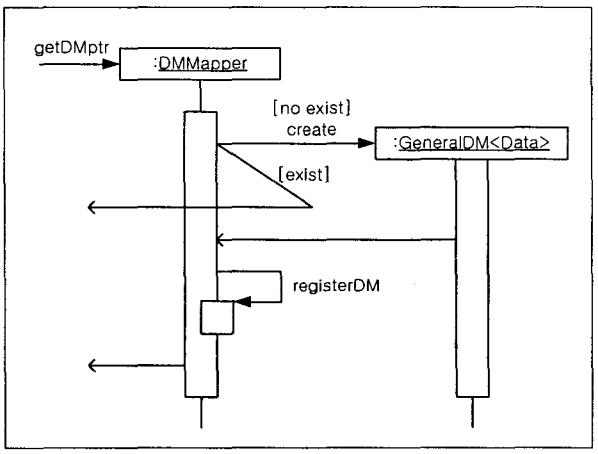

그림 8. DMMapper와 GeneralDM의 Sequence Diagram

이러한 DMMapper 객체는 C++의 기능 중 실행 중에 객체의 이름을 알 수 있도록 도와주는 기능(RTTI : Run-Time Type Information)을 이용하여 등록되는 DM이 관리하게될 데이터 객체의 종류를 인식하여 관리한다. 그림 8는 어플리케이션에서 DMMapper를 통하여 DM의 접근을 가능하게 하는 순서이다.

현재의 DMMapper 객체는 동일 프로세스에서 접근하는 것을 가정하여 설계 구현되었다. 그러나, ERP 기능 컴포넌트와 관리자 객체가 서로 다른 프로세스로 구성되는 경우에는 DMMapper를 이용하여 분산된 객체를 접근할 수 있도록 하여야 한다. 이러한 분산 환경하의 객체간의 통신은 CORBA와 같은 분산객체 미들웨어를 이용하여 가능하다. 표 6는 분산 환경하에서 DMMapper를 접근할 수 있도록 설정된 IDL이다. CORBA에는 이와 같은 기능을 담당하는 서비스(service)로 naming service와 trading service 등이 있다[12].

표 7. DMMapper IDL

```
#include <DataManager.idl>

interface DMMapper {
        void            registerDM(in DataManager DMptr);
        DataManager     getDMptr(in string DMName);
        void            delRegistry(DataManager DMptr);
};
```

4. 적용된 구조

이번 장에서는 3장에서 제안된 데이터 객체 계층을 여러 형태의 네트워크 구조에 적용해보고자 한다.

4.1 Client-Server 환경

클라이언트 서버 환경의 네트워크 구조로 ERP 시스템을 구축하는 경우에, 데이터 인터페이스는 기존의 ODBC와 JDBC 등의 규약에서 정하고 있는 기능으로도 해결할 수는 있다. 그러나 이러한 환경만을 전제로 작성된 어플리케이션은 추후 다른 형태의 네트워크 구조에서 재사용될 수 없다. 본 연구에서 제안하는 데이터 객체 계층의 경우에는 이러한 클라이언트 서버 환경에서 여러 방법으로 적용될 수 있다.

첫째, 어플리케이션과 데이터 객체 계층이 합쳐서 클라이언트가 되고, 기존의 ODBC 드라이버를 이용하여 서버인 데이터 저장소를 접근하는 구조 (그림 9-a).

둘째, 어플리케이션에 해당하는 기능 컴포넌트가 클라이언트가 되고, 데이터 객체 계층이 서버가 되는 구조 (그림 9-b).

셋째, 사용자 화면이 클라이언트가 되고, 기능 컴포넌트와 데이터 객체 계층, 그리고 데이터 저장소가 서버가 되는 구조 (그림 9-c).

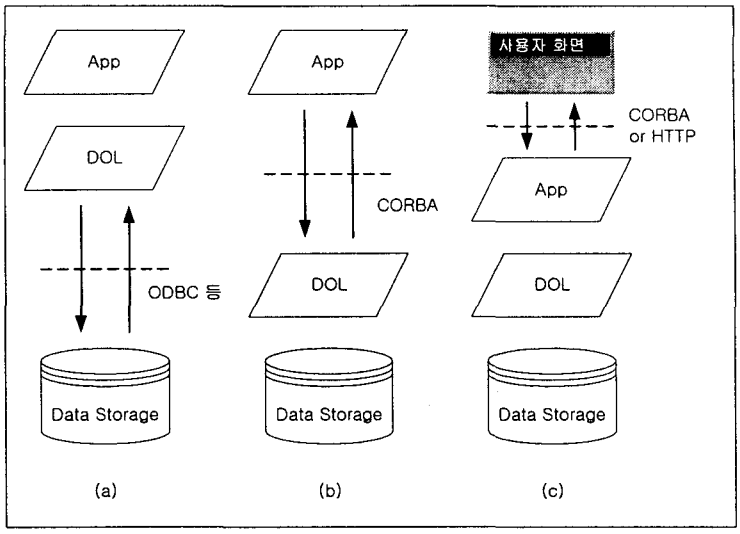

그림 9. 클라이언트 서버 구조

위와 같은 구조에서 첫 번째 구조에서는 데이터 저장소의 투명성을 위하여 ODBC 드라이버를 사용하여야 하나, 나머지 두 가지 구조에서는 어떠한 데이터 저장소 접근 기능 컴포넌트를 사용하여도 된다. 물론, 어플리케이션의 경우에는 어떠한 경우에도 변경사항은 없다.

4.2 Object Web 환경

기존의 클라이언트 서버 환경에서 웹 환경으로 변화하는 과정에서 기존의 ODBC로는 해결하지 못한 문제점을 해결하기 위하여 JDBC가 제안되었다. 그러나, JDBC의 경우에도 그 형태와 사용하는 네트워크 프로토콜에 따라서 4가지 형태의 JDBC 드라이버가 제안되어 사용되고 있다[14].

첫 번째 종류인 JDBC-ODBC 브리지(bridge)는 기존의 ODBC 드라이버를 사용하는 경우이므로, 클라이언트에 ODBC 드라이버와 데이터 저장소 접근 기능 컴포넌트가 다 올라와 있어야 하는 구조가 된다. 그러므로 이러한 종류의 JDBC 드라이버는 주로 회사 내부용 네트워크에서 사용되며, 웹-클라이언트-저장소의 3 단계 구조[13]로 사용된다. 두 번째 종류와 세 번째 종류의 JDBC 드라이버의 경우에는 클라이언트가 첫 번째 종류의 드라이버를 사용하는 경우보다 가벼워지므로, 2 단계 구조[13]로 사용될 수 있다. 마지막 종류인 native-protocol all-Java 드라이버의 경우는 중간에 임시 서버가 필요하므로 n 단계 구조로 설계되는 경우이다. 특히 이러한 경우 임시 서버가 웹서버인 경우에 적절한 경우이다.

본 연구에서 제안하는 데이터 객체 계층을 사용하는 경우에는 JDBC 드라이버의 어떠한 종류를 사용하더라도 사용 가능하다. 즉, JDBC 드라이버에 따라서 결정될 수 있는 네트워크 구조가 변경된다 하더라도 어플리케이션 코드는 변경하지 않아도 되는 투명성을 제공할 수 있는 것이다(그림 10).

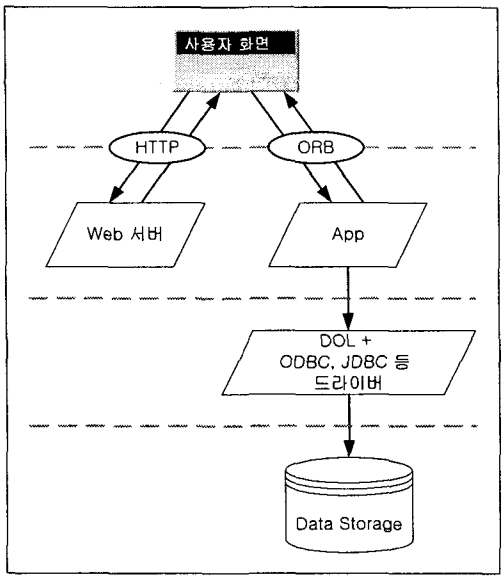

그림 10. 데이터 객체 계층을 사용한
n-tier 구조

5. 결론 및 추후 연구사항

본 연구에서 제안하는 데이터 객체 계층(DOL)은 ERP의 업무처리 기능에 해당하는 어플리케이션(기능 컴포넌트)이 데이터를 지속적으로 보관하기 위하여 데이터 저장소를 사용하는 경우, 어플리케이션과 데이터 저장소 사이의 구축되어 있는 네트워크 구조, 데이터 저장소의 종류와 위치, 그리고 구현 언어에 대한 투명성을 제공한다. 이러한 투명성의 증가는 ERP 시스템을 객체지향 기법을 이용하여 분석하고, 컴포넌트를 개발하는 과정에서 데이터 저장소와 연결되는 부분에 대해서는 변화될 상황에 상관없이 독립적으로 개발할 수 있는 장점이 있다. 이러한 데이터 저장소와 네트워크 구조, 구현언어에 대한 투명성은 기존의 ERP 패키지가 가지고 있던 폐쇄성을 극복하여 개방형 구조의 특징을 가질 수 있게 도와주며, ERP 개별 기능을 상업적으로 컴포넌트화 하여 독립적으로 제공할 수 있는 기반이 되기 때문에, 소프트웨어의 생산성을 높일 수 있을 것으로 생각된다. 또한 데이터 객체 계층은 ERP 기능 컴포넌트와는 다른 수평적 컴포넌트(horizontal component)로서 독자적인 기능을 수행한다.

본 연구에서 사용한 분산 객체 미들웨어인 CORBA는 COM/DCOM과 함께 분산 객체의 표준으로 자리잡아가기 때문에, 추후에 데이터 저장소의 변화, 구현언어의 변화 등에도 적절히 대응할 수 있을 것으로 보인다.

현재 분산 객체 미들웨어는 CORBA와 같은 기본적인 통신 미들웨어와 더불어, 트랜잭션 기능이나, 본 연구와 같은 지속성 기능 등이 추가된 복합 미들웨어로 발전해 나가고 있다. 본 연구에서는 이러한 미들웨어의 기능을 지적하였고, 그 중 지속성에 대하여 해결방법을 제시하고 있으나, 트랜잭션 처리 등 동시 제어 기능은 추후 연구 과제이다. 데이터 객체 계층에 이러한 미들웨어로서의 기능을 추가하면 ERP 시스템에 적합한 미들웨어로 발전할 수 있을 것으로 보인다.

참고문헌

[1] 권일명, 김창욱, 예성영, 전진, 김강호, 김성식, "데이터 객체의 일반화를 통한 공통 데이터 인터페이스의 구현,"「대한산업공학회/한국경영과학회 춘계공동학술대회 발표논문집」, 1998.

[2] 김창욱, 전진, 서동욱, 김성식, "Reusable and Tailorable Architecture for Enterprise Requirement Planning(ERP) Systems,"「대한산업공학회/한국경영과학회 춘계공동학술대회 발표논문집」, 1998.

[3] Arthur M. Keller, Richard Jensen, Shailesh Agarwal, "Persist Software : Bridging Object-Oriented Programming and Relational Database," *Proceedings of ACM SIGMOD International Conference on Management of Data*, Washing DC, pp.523-528, 1993.

[4] Bjarne Stroustrup, *The C++ Programming Language*, Addison Wesley, 1991.

[5] Chang-Ouk Kim, Jin Jun, Sung-Shick Kim, "Meta-Model Driven Collaborative Object Analysis Process for Production Information Domain", *submitted to International Journal of Computer Integrated Manufacturing*, 1998.

[6] Clemens Szyperski, *Component Software : Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[7] E. Gamma, R. Helm, R. Johnson, J. Vllissides, *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[8] Grady Booch, *Object-Oriented Analysis and design with Applications*, Addison Wesley, 1994.

[9] Ivar Jacobson, *Object-Oriented Software Engineering : A Use Case Driven Approach*, Addison Wesley, pp269-283, 1992.

[10] Mary E. S., Loomis, *Object Database : The Essentials*, Addison Wesley, pp75-86, 1995.

[11] Microsoft Corp., "Microsoft ODBC Overview," from Microsoft site, http://www.microsoft.com/data/reference/odbc2.htm.

[12] OMG, "CORBA 2.0/IIOP Specification", OMG formal/97-09-01, Sept. 1997, http://www.omg.org/corba/corbaiiop.htm.

[13] Robert Orfali, Dan Harkey, Jeri Edwards, *The Essential Client/Server Survival Guide*, John Wiley & Sons, 1996.

[14] Sun Microsystem, Inc., "The JDBC™ Database Access API," from Java/Sun site, http://java.sun.com/products/jdbc/.

[15] Søren Lauesen, "Real-Life Object-Oriented Systems", *IEEE Software*, pp76-83, March/April, 1998.

[16] Wilf Lalonde, John Pugh, "ODBMSs and Database Transparency", *Journal of Object Oriented Programming*, February 1994.